HIDE: An Infrastructure for Efficiently Protecting Information Leakage on the Address Bus

Xiaotong Zhuang Tao Zhang Santosh Pande

Georgia Institute of Technology
College of Computing
Atlanta, GA, 30332-0280
{xt2000, zhangtao, santosh}@cc.gatech.edu

ABSTRACT

XOM-based secure processor has recently been introduced as a mechanism to provide copy and tamper resistant execution. XOM provides support for encryption/decryption and integrity checking. However, neither XOM nor any other current approach adequately addresses the problem of information leakage via the address bus. This paper shows that without address bus protection, the XOM model is severely crippled. Two realistic attacks are shown and experiments show that 70% of the code might be cracked and sensitive data might be exposed leading to serious security breaches.

Although the problem of address bus leakage has been widely acknowledged both in industry and academia, no practical solution has ever been proposed that can provide an adequate security guarantee. The main reason is that the problem is very difficult to solve in practice due to severe performance degradation which accompanies most of the solutions. This paper presents an infrastructure called HIDE (Hardware-support for leakage-Immune Dynamic Execution) which provides a solution consisting of chunk-level protection with hardware support and a flexible interface which can be orchestrated through the proposed compiler optimization and user specifications that allow utilizing underlying hardware solution more efficiently to provide better security guarantees.

Our results show that protecting both data and code with a high level of security guarantee is possible with negligible performance penalty (1.3% slowdown).

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Miscellaneous;
K6. [**Management of Computing and Information Systems**]: Security and Protection.

General Terms: Security, Design, Performance.

Keywords: Secure Processor, Address Bus Leakage Protection.

ASPLOS'04, October 9–13, 2004, Boston, Massachusetts, USA.

1. INTRODUCTION

XOM-based secure architecture [1,2] has recently emerged as a model architectural support for copy and tamper resistant software. Under the XOM model, everything outside the processor chip is assumed to be insecure. The latest research on XOM architecture proposes OTP (one-time-pad) encryption/decryption schemes [3,4] to protect software confidentiality and a Merkle tree based scheme [5] to guarantee software integrity. Both schemes achieve the above security goals within a reasonable performance spectrum (a small performance degradation). Although the XOM model is successful in protecting the off-chip code and data using the encryption techniques (block cipher or stream cipher), it fails to protect the addresses on the address bus. In other words, the address sequence generated by an application may be exposed under the current XOM protection [3,4,5].

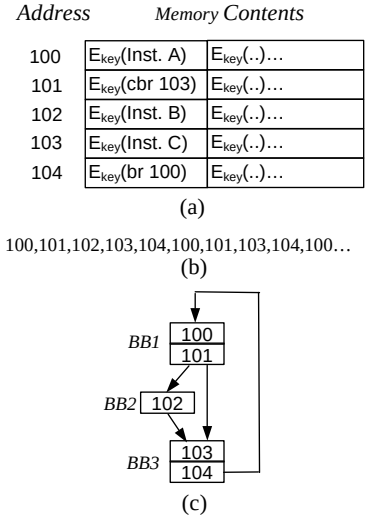


Figure 1. Control flow snooping.

In all previous XOM-based work, the following two questions are left unaddressed: 1) Even though off-chip memory contents are encrypted, can the exposed address sequence lead to security breaches? 2) If so, how should we prevent those at a reasonable cost?

Although the problem has been noticed in [1] and [6], they both leave it open. [1] poses it as an open problem, and [6] largely ignores it. In [5] it is shown that the detection of loops through the information leakage on address bus can become a starting point for the replay attack. In this paper, we point out that address bus protection is critical; otherwise control flow information might be exposed and severe security breaches might occur.

In Figure 1, we illustrate how control flow can be snooped by the attacker. Under the current XOM model, all 5 blocks of instructions are stored in an encrypted form, however authentic addresses are readily available on the bus. The attacker has no idea what the instructions are due to the encryption, however he can snoop on the bus and obtain a sequence of addresses (refer to Figure 1.b). From the address sequence, he can infer that the code is in a loop since addresses 100,101,103,104 appear repeatedly. Also, it is exposed that there is a conditional branch at address 101 because sometimes the control goes to 103 directly, sometimes it goes via 102 to 103. Therefore, by identifying recurring (block) addresses, the attacker can construct a *block level control flow graph (CFG)* as shown in Figure 1.c. Leakage of the control flow can severely jeopardize the encryption of code (it may be possible to crack as much as 70% of the encrypted code through a well devised attack – refer to Section 3) which is the basis of the XOM model. Apart from this, address sequence on the bus may lead to exposure of the critical data (such as the secret key) as well.

Regarding the second question, it may be noted that address bus protection is a much tougher problem than it might first appear. Both industry and academia are aware of the severity of information leakage through the address bus and have proposed solutions. DS5000 and DS5002FP are chips produced by Dallas Semiconductors [11], which are among the most widely used security devices in credit-card terminals, pay-TVs access control systems etc. The processor incorporates bus-encryption (actually, fixed address reordering together with some random accesses) and was described as the most secure processor currently available for commercial users. However, such protection can be easily invalidated (refer to [7] and related work). The only solution that completely avoids such information leakage is called Oblivious RAM (ORAM), which was proposed by Goldreich [9,10]. In his papers and patent, three schemes are proposed to ensure that the addresses on the bus are independent of the addresses issued by the application. Unfortunately, all three schemes are infeasible on real machines due to either significant slowdowns or resulting memory explosion. Therefore a practical solution for protecting information leakage through address bus is highly desirable and valuable for the viability of XOM-based secure processors.

This paper proposes such a solution with negligible overhead. Through hardware support and compiler optimizations, HIDE provides a very high level of security guarantee, which means that the information leakage via the address bus is largely prevented. Also, our infrastructure is highly flexible. It can easily incorporate programmers' specification of sensitive sections as well as some compiler optimizations.

The rest of the paper is organized as follows: Section 2 discusses the XOM model and attack model; Section 3 depicts possible attacks through the address bus leakage; Section 4 provides an overview of HIDE; Section 5 introduces basic concepts in HIDE; Section 6 presents chunk level protection; Section 7 shows how to protect more with HIDE plus; Section 8 talk about other considerations; Section 9 shows results; Section 10 and Section 11 provide related work and conclusion.

2. MODELS

XOM Model

Before elaborating on the address bus security vulnerability, we briefly introduce the XOM model [1]. The XOM model assumes that the only trusted hardware entity is the processor itself. Any other hardware components including the system bus and main memory are non-trusted since they are vulnerable to security attacks. Data and code are encrypted when they leave the processor and decrypted after they are fetched into the processor. Moreover, the operating system is also considered non-tamper resistant. Therefore, the XOM model provides mutual protection among processes running on the same processor. Later work on XOM [3,4,5] added integrity checks and performance enhancements. In fact, XOM is similar to previous models proposed in [8,10] and we can find real implementations of these models such as the DS5000/DS5002FP, smartcard chips, etc. in which the processor core is physically shielded and code and data are maintained in the encrypted outside the chip.

Attack Model

Protecting information leakage implies stopping the attacker from getting any useful information related to the intellectual property of the software; such protection is more general than the copy protection (i.e., protecting someone from making illegal copies). For example, consider a scenario in which company A and company B are developing similar software. Company A succeeds first and releases the software in an encrypted format, which (presumably) is execute-only on XOM machines. However, company B now has complete access to the software bundled with the XOM machine on the market. Company B can experiment with it extensively, e.g., feed the program with different inputs, execute that software together with their own software on the same machine or even with a manipulated OS. Moreover, company B is already an expert in developing similar software, therefore it only wants to understand a few critical parts of company A's software. As shown later, techniques such as CFG matching can distill important information from the unprotected address bus. Although this scenario is not as straightforward and prevalent as software piracy by end users, it has been a major concern of software companies (the recent Linux/SCO-Unix lawsuit is an example).

As a matter of fact, these kinds of attacks, i.e., circumventing the encryption scheme indirectly through information leakage are well-known in the security domain as *side-channel attacks*. In reality, there have been many successful stories [14,15,18,19] to obtain critical information from a secure chip such as a smartcard, by monitoring the timing [14], power [15] or electromagnetic differences [16] from outside the chip.

3. ATTACKS VIA CONTROL FLOW SNOOPING ON BUS

We now illustrate two attacks that are possible through the control flow information leaked on the address bus. Notice that, the example in Figure 1 assumes that there is no "noise" in the addresses seen on the bus (i.e., all the addresses are leaked), but in practice, branches within a block are hidden, a cache can hide many accesses, etc. This might lead to less than full leakage but could still be quite damaging. We first assume that there is no noise, and in Section 3.3, we will address the noise issues.

3.1 Reuse Code Identification

Due to the following two facts, leaking the CFG information can result in the complete exposure of reuse code and severely disrupt code encryption.

Software Reuse and Binary-Level Similarity

With the ever-increasing amount of legacy code and time-to-market pressure, software development relies more and more on reusing existing modules or on pre-built libraries from other companies or oftentimes from the public domain. For example, many classic algorithms have their standard and/or non-standard open-source implementations online ready for reuse. Moreover, most compiler and development tool chains are provided by a few 3rd party name-brand vendors that can lead to a high binary-level similarity once the source code is reused. We measured the full set of SPEC 2000 Alpha binaries to find out the percentage of code that is reused from the standard C library on Alpha. As shown in Figure 2, the reuse percentage can be very high for some benchmarks like mcf (88%) and bzip2 (66%). On average, 39% of the code at binary level is due to libraries. A recent study [22] shows that nowadays, up to 70% of the code in industry software is reuse code. Given such a high amount of reuse code, the question is: Can it be discovered? The answer is yes. Address bus leakage allows building a CFG and CFGs serve as unique fingerprints of underlying code leading to such a discovery. Once such reuse is discovered, the attacker knows the key underlying algorithm or the intellectual property (IP).

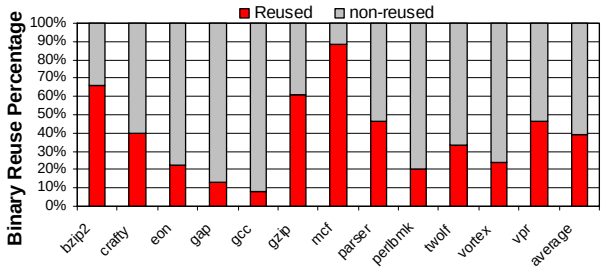


Figure 2. Binary reuse percentage for SPEC2000.

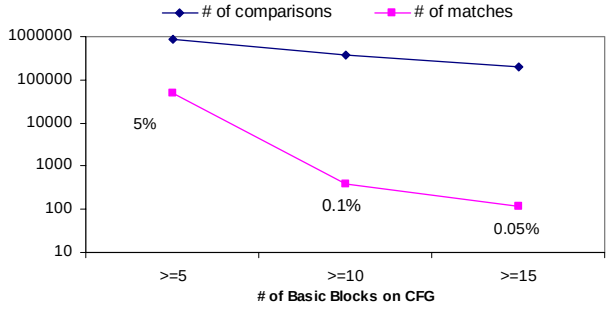


Figure 3. Isomorphic CFG pairs in the standard C library.

CFG-Fingerprint of Algorithm

In order to determine the uniqueness of CFGs we did another study and found that they indeed serve as unique fingerprints due to the following intuitive arguments. CFGs are made of a few basic blocks. It is widely known that the average length of basic blocks is only 6 to 10 instructions for integer programs and a large number of instructions are branches (around 12%). Conceivably, as long as the algorithms are reasonably complex, the chance of forming the same CFG is slim since they would have a good number of basic blocks and quite a few potential control flow graphs are possible with a given number of basic blocks. As an experiment, we built the CFGs for various block cipher algorithms such as DES, MARS, Rijndael, RC6, and found out their CFGs are significantly different. In Figure 3, we investigate the similarity of CFGs in the standard C library of the Alpha compiler. There are 1334 procedures in the library file libc.a, with reasonable size (at least 5 basic blocks). We built the CFGs for all these procedures in which each basic block is abstracted as a node (which in fact increases the chances of two CFGs being similar). We run the famous graph isomorphism algorithm by Ullman [12] (we reuse the graph matching library developed by Univ. of Naples [13]) between all possible pairs of graphs. In Figure 3, the results show that only 5% of the comparisons find that the two graphs match. If we ignore the CFGs with less than 10 basic blocks, only 0.1% match. Finally, if we ignore the CFGs with less than 15 basic blocks, only 0.05% match. This study shows that each CFG can serve as a distinct fingerprint for a reasonable-sized code. Therefore, if the programmer reuses a procedure in the library with 10 or more basic blocks, the reuse is almost doomed to be found out by the attacker due to its distinctive fingerprint (assuming he can construct the CFG using address bus leakage). Notice that, this estimation is conservative due to our abstraction of the CFGs that ignores sizes of individual basic blocks; otherwise the number of matches would decrease further. Even if some matches occur, the attacker can still narrow down his search to a few possible procedures that might be reused.

Given sufficient amount time to experiment with the code, most CFG edges could be exposed. Theoretically only dead code is not executed. Even if only partial CFG can be identified with subgraph matching algorithms [12,13], we can still largely detect the reuses. It is easy to show that the number of legitimate CFG graphs grows exponentially with the number of basic blocks in the CFG; therefore hiding big reuse code is almost impossible. From the prior discussion, the CFG, as a matter of fact, can be regarded as an algorithm's fingerprint.

Based on the two facts described above, it is quite possible that an attacker can identify the reuse components in a program given its CFG. He can collect the CFGs of all procedures in the standard libraries, or for publicly available source code, compile them with a name-brand 3rd party compiler and build the CFGs. By graph matching the program's CFGs with his collection, the attacker can nail down the reuse parts. This not only exposes the reuse code in its entirety, but also helps the attacker in other aspects: 1) A bunch of plaintext/ciphertext pairs for the reuse code are identified. If the hardware cannot afford integrity check due to its prohibitive performance and memory space overhead [5], the attacker might construct a program to read out other code such as in [7]. 2) More critically, in some cases critical data could be leaked due to the discovery of re-use code. In the next subsection, we will show how critical data can be found out in some cases. 3) By watching the interaction between reuse code and the programmer's own code like calling sequence, parameters, the attacker can learn more about programmer's own code.

3.2 Critical Data Leakage via Value-dependent Conditional Branches

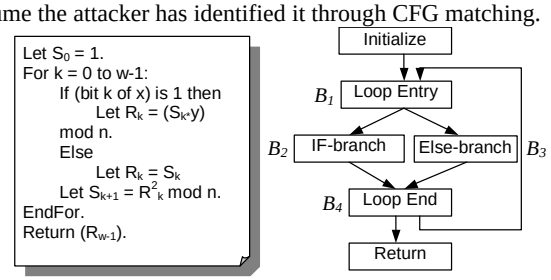
Apart from the above potential problem of revealing IP, CFG matching can also potentially compromise a secret key and leak sensitive data.

All conditional branches (around 80% among all branches) make comparison between two values and then decide which path to take. Therefore the control flow information can leak important information about the values being compared. The following example assumes that the algorithm used is known beforehand

(most security systems assume the cryptographic algorithms used are known to the attacker) or has been detected by CFG matching. It demonstrates how the critical data (secret key in this case) is revealed.

Example

Diffie-Hellman and RSA private-key operations consist of computing $R = y^x \bmod n$, where the attacker's goal is to find x, the secret key. To show the problem easily, we assume that the implementation uses the simple modular exponentiation algorithm in Figure 4.a, which computes $R = y^x \bmod n$, where x is w bits long. The algorithm is widely used, therefore we can reasonably assume the attacker has identified it through CFG matching.



(a) (b)

Figure 4. Modular exponentiation algorithm.

The corresponding CFG for this small piece of code is shown in Figure 4.b. From Figure 4.a, we can easily find that inside the loop body if the current examined bit of x is 1, IF-branch is executed, otherwise Else-branch is executed. We assume IF-branch code resides at address B_2 and Else-branch code resides at address B_3 (B_2 and B_3 are different). The secure processor must behave as follows: if the current examined bit of x is 1, then fetch the IF-branch code at B_2, otherwise, fetch the Else-branch code at B_3. This results in a sequence of addresses for B_2 or B_3 showing up on the address bus correspondingly. By monitoring the address bus and capturing the addresses transmitted, the attacker can guess whether the respective bits of x are 0's or 1's and get the secret key x. Even if he cannot distinguish between IF-branch and Else-branch, the information on the address bus leaves only two possible values of x to guess (the correct key or its complement).

This example tells us that if the conditional branch is known to the attacker, the direction of the execution path after the branch can expose the outcome of the comparison which might be helpful in determining or narrowing down the values involved. Such security sensitive conditional branches are widely seen in compression and encryption algorithms. Leakage similar to this has led to timing attack [14] which is however more complicated.

Notice that missing several rounds of the for-loop can hide part of the secret key, but still help the attacker substantially to narrow down his search space. It is well known in the security domain, 64-bit encryption has much less strength than 128-bit encryption. If the attacker can capture half of the for loop, his search space will be cut from $2^{|x|}$ to $2^{|x|/2}$, which is $2^{|x|/2}$ times faster.

3.3 Noise: Blocking, Caching

Most side-channel attacks have to deal with noise that leads to inaccuracy: Timing attacks suffer from varied computation time of instructions, power attacks must count for the power consumption of other components inside the chip. However bus snooping is actually more accurate than other side-channel attacks and it is very easy to setup [7,21]. Here we discuss different types of "noises" that can affect control flow snooping and how the attacker may get around them.

Blocking

Cache misses are typically addressed (and accessed) at block boundaries; thus, the addresses on the bus are block addresses. Actually, both attacks we mentioned above only rely on the detection of branches. Given each block contains very few instructions (8 on Alpha), it does not affect the attacks much. For reuse code identification, we tried to build block-level CFG, i.e., every block becomes a node and edges indicate possible execution paths between blocks. We found that typically block-level CFGs contain about 25% lesser edges than the regular CFGs. Graph matching the block-level CFG shows results close to those in Figure 3 with negligible changes. To find out how block size affects CFG matching, we list in Table 1 the percentage of matched block level CFGs when the block size equals 32B, 64B and 128B. It is interesting to see that if we ignore block level CFGs with less than 10 or 15 nodes, the increase in block size does not necessarily reduce the number of matches. This is probably because most non-reuse procedures are larger than the reuse procedures in the library as we have observed. Thus, this experiment shows that larger block size does not affect CFG matching much.

Table 1. Isomorphic block level CFGs with different block sizes.

	32B	64B	128B
≥5	5%	4%	3.3%
≥10	0.1%	0.19%	0.1%
≥15	0.05%	0.04%	0.05%

Caches

Modern processor typically consist of large on-chip caches which might lead to small miss ratios and very few addresses exposed on the address bus. However, it does not help due to the following reasons. (1) Since the cache is a shared resource among all processes running on the processor and as in the previous papers, XOM assumes that the OS is not secure. It is very easy for the attacker to manipulate the OS so that the cache gets flushed upon a context switch; alternatively the attacker can ascertain that his own process fills and occupies most cache space before switching to the process being attacked. In this manner, all memory accesses are exposed directly on the address bus due to compulsory misses. (2) Even if only one process is running, many processors have a unified L2 or L3 cache for both code and data. If the program's working set can be affected by inputs, the attacker may intentionally increase the working set size causing more instruction misses. (3) Generally, caching is not predictable, especially in multi-tasking environment. Different parts of the control flow can be leaked during different runs. It is possible that the attacker can finally get the whole picture. (4) For low-end systems, on-chip caches are typically small. (5) The cache may be disabled on some machines.

As an experiment, we tried to flush the cache at random moments, and collected 4 block addresses immediately after the flush. After sufficient number of runs, we found that over 95% of edges on the block-level CFG were exposed. In addition, as mentioned before, even if the control flow can be partly masked, information still leaks to some extent since subgraph matching can match partial CFGs. Partial execution path can still be used to prune the searching space for critical data.

The two attacks we showed above are very simple compared to some of the side-channel attacks, which involve sophisticated mathematical and statistical analyses. This indicates address bus information leakage is relatively easier to exploit as well as more damaging and is harder to prevent. With more advanced analyses, more information leakage could result.

3.4 Data Address Protection

Finally, accesses to the data segment can expose control flow as well. For example, in Figure 4.a, if y is accessed, we will know that the If-branch is taken. Therefore, data address protection is equally important. However, this could induce a big overhead since the size of the data segment can be much bigger than the code size.

4. HIDE—PRELIMINARIES

HIDE stands for *Hardware-support for leakage-Immune Dynamic Execution*. HIDE provides an infrastructure for preventing information leakage on the address bus involving both an micro-architecture as well as a compiler. The basic idea behind HIDE is to break the correlation between repeated memory addresses. This is achieved by permuting the address space at suitable intervals during the execution.

In this section, we first introduce basic concepts and components of HIDE. We then talk about what kind of address sequence should appear on the address bus to avoid information leakage and the two hardware components: the hide cache and the permutation unit.

4.1 Probabilistically Fixed Address sequence

To hide the address sequence on the bus, a naïve but fully secure approach is to establish a *fixed address sequence* that does not change throughout the execution [10]. For example, the processor can read and write each block in the whole memory repeatedly from the lowest address to the highest address in a fixed order. If a block is required by the program or is to be written out, the processor must wait till the "repeated read/write sequence" reaches that block. Obviously, this naïve approach can cause significant slowdown, given the memory space is big and one round of accesses can take tremendous amount of time.

As introduced by [10], we can also construct an address sequence with addresses conforming to a fixed probabilistic distribution. In other words, if the addresses seen on the bus are random variables conforming to a fixed distribution, it still exposes no information about which addresses are actually accessed by the processor.

Original Address (Sequence)*:* The address (sequence) issued by the processor.

Actual Address (Sequence)*:* The address (sequence) that actually appears on the address bus.

Probabilistically Fixed Address Sequence*:* A kind of actual address sequence in which actual addresses follow a fixed probabilistic distribution.

The following lemma gives one such probabilistically fixed address sequences.

LEMMA 1: A memory space of size M is randomly permuted repeatedly; assume a block originally at address T is relocated to $P_k(T)$ after the k^{th} permutations. If between the k^{th} and $(k+1)^{th}$ permutation, the processor accesses original addresses T_1, T_2,…$T_{n(k)}$, and these addresses are all different, then the address sequence on the bus is probabilistically fixed.

REMARK: Lemma 1 says that, between two permutations, all original addresses should be different, or we should randomly permute the memory space before the same original address is issued again. Notice that since different blocks cannot be permuted to the same location (mapping is one-to-one), all actual addresses between two permutations are different too.

PROOF: Lemma 1 is derived from [10]. We can prove this lemma as follows. Since permutation P_k is completely random and one-to-one, for two different original addresses T_i and T_j , $P_k(T_i)$ and $P_k(T_j)$ are different (two different addresses cannot be permuted to the same place) and are independent random variables. Therefore the addresses in the sequence based on the same permutation P_k, i.e. $P_k(T_1)$, $P_k(T_2)$… $P_k(T_{n(k)})$ are independent to each other. Similarly, since any two permutations P_k and P_l are random and independent, $P_k(T_i)$ and $P_l(T_j)$ are always independently distributed. Thus the addresses on the bus are all independently distributed variables. Notice that, between two permutations, original addresses must be distinct, otherwise we will see the same actual address recurring on the bus (this is because the same permutation must permute the same original address to the same actual address). From control flow point of view as mentioned in Figure 1, recurring addresses help the attacker to identify loops and branches. For a better understanding of Lemma 1, we will give an example during the discussion of the *hide cache*.

4.2 Hide Cache

To fulfill Lemma 1, intuitively we must "remember" the original addresses that have been issued by the processor after the previous permutation. Before an original address recurs, we need to permute the memory space again. In other words, we must remember the original address sequence to detect recurrence of an address. It is clear that if we "remember" only a small number of original addresses, the memory space must be permuted more frequently. On the other hand, if we "remember" a lot of original addresses, extra space is required to store them and more latency is incurred to check if a new original address has been issued before.

The *square root algorithm* in the ORAM paper [10] stores all such original addresses in an off chip memory called *shelter buffer*. The size of the shelter buffer is the square root of the memory space being protected. However, during each access, the processor must read the entire shelter buffer to check if the address has been accessed. Since the shelter buffer is in the insecure memory, the processor must read the entire shelter buffer such that the attacker cannot tell whether or where the access has hit in the shelter buffer.

With large on-chip space available on modern processor, we may want to move the shelter buffer on-chip. However, it is still space inefficient to occupy a separate on-chip area for this purpose. Given caches are readily available to store memory blocks accessed before, in this work we propose the *hide cache*; which adds address bus protection on top of a normal cache to achieve a low space and performance overhead solution.

Hide Cache*:* A cache same as a normal cache except that blocks fetched after the previous permutation are all locked i.e. they cannot be replaced until the memory space they belong to is

permuted again. Also, blocks that are dirty after the previous permutation must be held from the write back until the next permutation.

How the Hide Cache Works

In a hide cache, we intentionally lock all blocks that are fetched after the previous permutation. Therefore accesses to the same original address between two permutations always hit in the cache without going out to the memory. Similarly, blocks that become dirty after the previous permutation are locked as well. If such dirty blocks are allowed to be written back, there could be read accesses to the same address later causing the same address to appear on the bus again; thus, such blocks must be locked as well. If a block is locked, it cannot be evicted. When the memory space is permuted again, all blocks belonging to that memory space are unlocked.

Next we show an example in Figure 5, which helps to understand Lemma 1 and the hide cache. In Figure 5.a, we assume that the cache is 2-set 2-way. Figure 5.b shows the original address sequence borrowed from Figure 1. The memory space contains 5 blocks as shown in Figure 5.c. Notice that all blocks are initially permuted randomly after they are loaded from the disk to the memory. The initial random permutation prevents the attacker from correlating information across different runs (since random permutations are done before code and data are initially loaded). We also assume all accesses are read accesses in this example. If blocks are not locked and permuted after the initial permutation, we will observe the cache contents and actual access sequence as illustrated in Figure 5.d. On the left side of Figure 5.d, we show the status of the cache after 4 fetches. All addresses shown inside the blocks are the original addresses. Since the four addresses are all different, they are loaded from memory due to compulsory misses. Also, since blocks are already permuted, the actual address sequence on the bus is 102,100,104,101,103 instead of 100,101,102,103,104 after 5 accesses. If the blocks were not locked (as in normal caches) we will see 102 (which corresponds to the original address 100) appearing again. This means that even if the block 100 is randomly permuted to address 102, its recurrence can still be detected on the bus. Thus, a normal cache cannot hide such recurrence. With hide cache—as shown in Figure 5.e—the 2nd permutation is triggered before the 5th access, because all blocks in set 0 are locked and we cannot evict a locked block. Notice that if a locked block is evicted, we lose tracking of the block. It might be read in again causing the same address to appear on the bus; or if the block is locked because it is dirty, evicting the block will incur a writeback immediately causing re-appearance of its address on the bus. Thus, a permutation *must* be conducted to unlock blocks when all blocks are locked in a set. From the address sequence at the lower right part of Figure 5.e, we can observe that after the 2nd permutation, the original address 100 is now 104. Since the two permutations are random and independent, the attacker only observes random numbers on the bus. Also, recurring original addresses become different random numbers after the 2nd permutation. By locking the blocks, we are sure that the same address does not appear again on the bus *before* another permutation. For example, if we were to access block 104 again, it is in the cache; therefore no access goes out on the bus due to a hit. This scheme not only meets the requirement of Lemma 1, but also preserves the functionality of a cache.

Another observation in Figure 5.e is that in set 1, both blocks are unlocked after the second permutation. Now they behave like normal cache blocks and can be evicted if necessary. Since their mapping and addresses have been changed (their original address is now mapped to a different one) during the second permutation, they can be safely evicted. After the blocks are evicted, they will be locked the next time they are fetched in. Finally, in this example, we show how latency might be incurred. When all blocks are locked in a set, we must permute to unlock at least one block before a new block can get in and replace the unlocked one. Since permutation takes a long time, it is not wise to permute at the last minute. We thus permute and unlock some blocks before all entries are locked (this issue will be addressed in detail shortly).

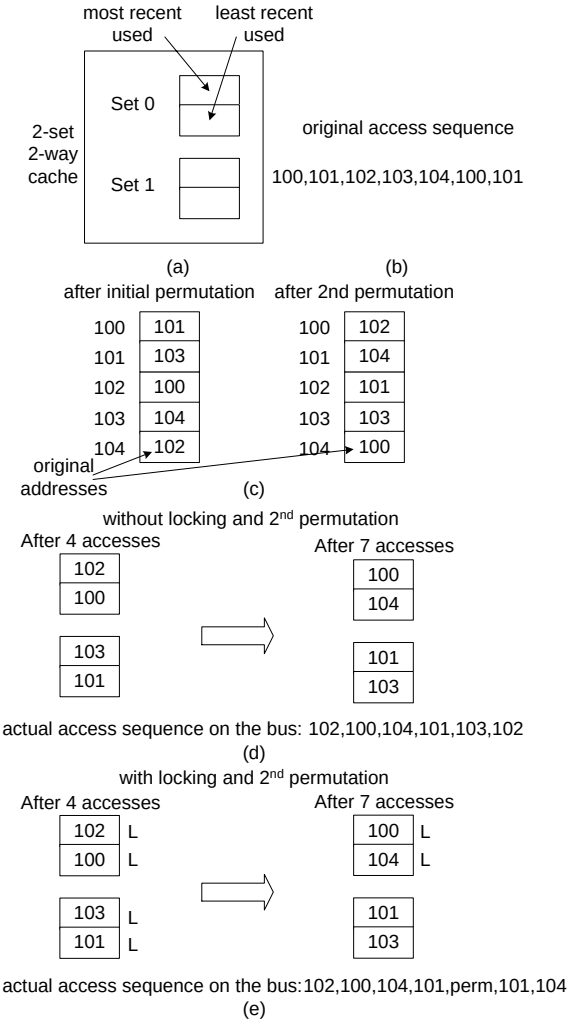


Figure 5. Example for Lemma 1 and hide cache.

Implementation of the Hide Cache

We list all operations in Table 2, assuming LRU is the original replacement policy. Notice that blocks are still ordered and updated according to the LRU policy, except that when evicting a block, one should choose the least recent used block among the unlocked blocks.

In our implementation, we use a bitmap (separately stored) to record whether a block is locked or not, i.e. each bit represents one block. After the permutation, the whole bitmap is cleared.

Latency Hiding via Fetch Buffer and Pre-permutation

Notice that locking blocks, i.e. setting the bits in bitmap is not on the critical path since it can be conducted after the cache access. For cache misses, for each block in the set we need to determine whether it is locked or not. This involves reading several bits in the bitmap. Fortunately it is not on the critical path either. The missed block can be fetched in parallel when we access the bitmap. Even if the missed block is fetched faster than the bitmap accesses, we can simply put the block in a *fetch buffer* and let the cache access return first. Once we determine all unlocked blocks and find the LRU block among them, the block in the fetch buffer can now be moved into the cache to replace the LRU block. In this way, all the operations for the hide cache can have the same latency as a normal cache.

Table 2. Operations on hide cache.

Read and hit	Update LRU order of blocks
Write and hit	Set dirty bit and lock the block, update LRU order of blocks
Cache miss, fetch a new block into the cache	Picked the LRU block among all unlocked blocks in the same set. Fetch and lock the new block. Set dirty bit for write access. If all blocks are locked, put it to fetch buffer.
A memory space is permutated	All blocks belonging to that memory space in cache are unlocked

However, the performance of a hide cache could still be worse than a normal cache because 1) A locked block may otherwise be replaced when it becomes least recent used. 2) Once all blocks in a set are locked, a new block cannot be fetched into the set unless we permute and unlock some of the blocks. Due to the long latency of permutation, it may happen that a set is fully locked before the permutation releases a locked block in the set leading to stalls.

We propose *pre-permutation* to solve the above two problems. Pre-permutation attempts to start permutation before all blocks are locked. In our design, we start permutation when half of the blocks in a set are locked. Pre-permutation increases the chance that a block is unlocked before it becomes LRU, and greatly reduces the possibility that all the blocks in a set are locked when new blocks need to be fetched in. Even if the permutation does not complete in time, we can put newly fetched blocks in the fetch buffer, until the permutation completes and unlocks blocks for replacement. These latency hiding techniques successfully cut down the performance loss as shown in our results.

4.3 The Permutation Unit

The permutation units randomly permutes the memory space. It should avoid exposing the correlation between any block's old and new locations. We show its pseudo-code in Figure 6. A memory space with M blocks is to be permuted using an on-chip space with P blocks called *out_buffer*. In addition, a *permutation vector (pv)* consisting of a random permutation of numbers from 1 to M is generated and stored on-chip.

CASE I: If M is less than or equal to P, we only need to sequentially read the M blocks once. After reading in block number s, we put it to *out_buffer*[*pv*[s]]. Finally, *out_buffer* is written out sequentially to the original memory space. Since the attacker only sees one sequential read and one sequential write to all blocks and everything is re-encrypted, he cannot build any correlation between blocks' old and new locations.

CASE II: If M is larger than P, without loss of generality, let $M=k*P$, where k is an integer larger than 1. We split the M block memory into k equal-size partitions. During an iteration s, all blocks destined to partition s are permuted and put in the *out_buffer*. At the end of iteration s, the *out_buffer* is written out to a temporary memory space as a new partition s. Upon finishing all permutations, we overwrite the original memory space with blocks in the temporary memory space. Overall, the M blocks are read k+1 times and written 2 times. Alternatively, we can change the page table such that the temporarily memory space is mapped as the original memory space, which avoids copying from *temp_mem* to *mem*.

Although the size of the *out_buffer*, i.e. P cannot be very large, the permutation unit can permute very large memory space, i.e. M can be large. In this algorithm, the size of *pv* still depends on M, however *pv* is actually very small because it only stores a short integer for each block (8 bits for 8K pages with 32B block).

The pseudo-code in Figure 6 does not show the algorithm to generate a random permutation of numbers from 1 to M. We follow the shuffle algorithm in [23], which requires M swaps to generate a complete random permutation as long as a hardware-based true random number generator is available. Finally the time for random permutation generation can be completely masked when the permutation unit reads the memory space.

```
DATA STRUCTURE:
//memory space to be permuted
block  mem[1..M]
//temporary space in memory
block  temp_mem[1..M]
//permutation vector, on-chip
Int      pv[1..M]
//output buffer, on-chip
block out_buffer[1..P]
```

```
M≤P
pv[1..M]⇐a random permutation of numbers from 1 to M;
for s=1 to M do
  out_buffer[pv[s]] ⇐mem[s]
endfor
mem[1..M] ⇐re-encrypt(out_buffer[1..M])
```

```
M=k*P
pv[1..M] ⇐a random permutation of numbers from 1 to M;
for s=0 to k-1 do
  for t=1 to M do
    read mem[t];
    if s*P<pv[t] ≤(s+1)*P then out_buffer[pv[t]-s*P] ⇐mem[t]
  endfor
  temp_mem[(s*P+1)..(s+1)*P] ⇐re-encrypt(out_buffer[1..P])
endfor
mem[1..M] ⇐temp_mem[1..M]
```

Figure 6. Pseudo-code for the permutation unit.

There might be pending accesses to the memory space being permuted. We can either issue the access to memory if the block has not been read-in (Lemma 1 is still enforced, because we do not unlock until the permutation finishes). If the block is in *out_buffer*, it is fetched immediately. If the block has been written out, we can still fetch it from the memory, but need to mark that it will remain locked after the permutation.

Permutation Overhead

With pre-permutation, permutation is normally not on the critical path; critical reads by the processor are always given

higher priority than the permutation traffic. Notice that although permutation is the main source of bus traffic increase, such traffic is very regular and predictable and therefore can be easily pipelined. In addition, we can take advantage of memory banking and parallelizing memory accesses to different banks. If a normal access is going to access the chunk that is being permutated (this should rarely happen, since a chunk only takes a small portion of the address space), we can first try to locate it in the *out_buffer* if that block has been read into the permutation chip. To amortize the initial overhead for each bus transaction (which could take the majority of the access time if only a small number of bytes are transferred in each transaction) we should read/write many consecutively located memory blocks during each transaction. Finally, we can completely offload the permutation traffic from the front-side bus with a separate *permutation chip* (or combined with the memory controller). There is communication between the processor chip and the permutation chip, e.g. the processor chip should send/receive data to the permutation chip if the address falls in the chunk that is being permuted. Also the permutation chip should return the new mapping once a permutation has finished. Obviously, such communication should be in encrypted form to be immune from bus tapping and the amount of traffic is actually much less.

We now show how we put together the HIDE cache and permutation unit to offer chunk level protection.

5. HIDE AT CHUNK-LEVEL

This section gives the hardware infrastructure of HIDE, which provides chunk-level protection with simple interfaces. Here *a chunk is defined as one or more pages that are protected and permuted together*.

Protecting a large piece of memory is prohibitive due to the high permutation cost, esp. when the *out_buffer* cannot hold the entire piece of memory, we must access the memory multiple times. The goal of chunk level protection is to limit the size of the permutation. At chunk-level the permutation unit only permutes all the blocks *within* a chunk. Once a chunk is permuted, all cache blocks in that chunk are unlocked.

We can split an address sequence into a series of transitions from address to address, e.g. the address sequence in Figure 1 has transitions like: 100→101, 101→102, 102→103…If the transition is between two addresses in the same chunk, we call it *intra-chunk transition*, otherwise it is *inter-chunk transition*. Since all intra-chunk transitions are protected with chunk-level protection (blocks within a chunk undergo permutation), the percentage of intra-chunk transition among all transitions is called *transition coverage*, which is a good indication of how well the address sequence is protected and *the level of security guarantee* we can provide.

We found chunk-level protection is powerful. As observed from our benchmarks, even with the smallest chunk size i.e. a page, over 75% of the transitions are intra-chunk. Given that not all memory contents are security sensitive, protecting chunks of a reasonable size should suffice if we can slightly narrow down the protection domain with either compiler analyses or user specifications as shown in Section 6. Besides, chunk-level protection is flexible. Since our infrastructure supports chunks with different sizes, the user can choose to protect some memory space in big chunks and some in small chunks. Building chunks on top of pages facilitates the implementation, since pages are supported by both the hardware and OS.

Figure 7 shows the overall hardware structure to provide chunk level protection. The L2 cache is now a hide cache. The fetch buffer and the permutation unit were introduced in Sections 4.2 and 4.3. The controller coordinates all components.

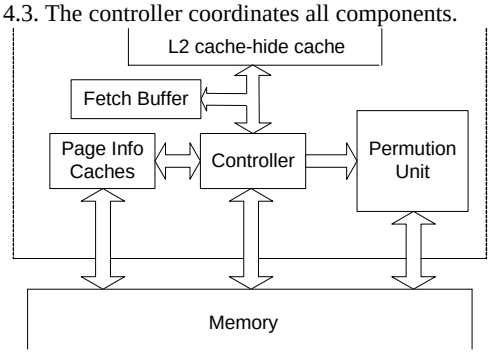


Figure 7. Hardware flowgraph.

Before addressing the page info cache, we first describe the *page info record*, which contains 7 fields for each page. Page info records store extra information for the L2 cache so that it can function as a hide cache. As shown below, the *if_hide* field indicates whether this page should be protected. Blocks in a protected page are permuted together with other pages in the same chunk and accesses must go through address translation to reach the new locations. The second field is the page number in the virtual address space. The next two fields specify the chunk this page belongs to. Notice that each chunk must take a contiguous piece of memory in the virtual address space to facilitate compiler optimization and user specification. In other words, a chunk must take several consecutive pages in the virtual address space; *begin_virtual_page#* and *chunk_size* (number of pages in a chunk) uniquely define a chunk.

```
Page Info Record:
boolean     if_hide;
int         virtual_page#
int         begin_virtual_page#;
int         chunk_size;
int         num_in_cache;
boolean     lock_bitmap[num_blk];
blk_addr_t  translation_table[num_blk];
```

num_in_cache counts the number of blocks of the chunk that are locked in the *hide cache*. For chunks with multiple pages, only the first page's info record stores such information. This field is used for pre-permutation described in Section 4.2. When half of the blocks in a set are locked, for each locked block we find out the percentage of locked blocks in their chunks. The one with the highest percentage is chosen to be permuted. The *lock_bitmap* field contains *num_blk* bits, where *num_blk* is the number of blocks in a page. Each bit indicates if the block is in the cache and is locked. Finally, the translation table translates each block to its new block address after a permutation. The translation table is updated by the permutation unit after each permutation. For chunks with multiple pages, the *blk_addr_t* includes a *page_ID* $\in$[0,*chunk_size*) to indicate which page in the chunk this block is permuted to i.e. *begin_virtual_page# +page_ID*.

The size of the page info record is small compared with the size of a page. For 8KB page size on Alpha, it only adds 3.5% space overhead in memory. However, for big chunks, the translation table takes more space due to the bigger *page_ID* field.

Page info records are stored separately in a dedicated memory space that can only be accessed by the hardware. There is

one page info record for each physical page and the page info record accompanies the page even when it is swapped out. For each block, the hardware can find the corresponding page info record via its physical page number. To speedup the access to the page info records, we put a page info cache on chip. Due to the small size of the page info records, a cache of 8~16KB is typically enough to achieve very high hit rate. However, since page info records are stored in memory, accesses to this data may leak information on the address bus although they are encrypted under the XOM model. Perceivably, such information leakage is indirect and also very limited (only 3.5% in size). For complete security, we can build several layers of protection, i.e., the first layer page info cache becomes a hide cache which protects the first layer page info records for pages used by the program. Since the first layer page info cache is a hide cache, we need second layer page info records for this hide cache and the pages taken by the first layer page info records. Obviously, the sizes of the page info cache and page info records decrease exponentially. At layer 3, the page info records are typically small enough to be stored on-chip.

All pages in a chunk should be permuted together; thus they must be swapped in and out together to avoid disk access latency. In our implementation, these requirements are conveyed to the OS when a chunk is created. Even if the OS is malicious, it cannot change the page info records, and while it may violate the swapping requirements, these violations will be detected by hardware. Swapping pages at the level of chunks can have the same effect as increasing the page size, causing extra memory pressure when some pages are swapped without real accesses to them. However, we conjecture that a chunk contains pages among which there are many transitions. These pages very likely in the same working set and are accessed in the same period. Thus, swapping the pages together should have minor performance penalties. In fact, it may even have the effect of improving performance because of prefetching.

Finally context switches should not affect the hide cache, since it works at physical page level, i.e., page info records are designed for physical pages and are addressed with physical page numbers. It works in tandem with a L2 hide cache where all addresses are physical addresses.

Interface to the Application

Chunks can be specified statically in the code or at runtime with special instructions. In the former case, such information is inserted in the header of the binary code telling which chunks should be created initially. Upon loading a chunk, the hardware initializes the fields in the page info record and performs a permutation before loading it into the memory. The initial permutation prevents the attacker from gaining information across different runs of the program. At run time, we can use 3 instructions to manage chunks; their syntax and operational semantics are as follows:

♦*hide_chunk (begin_virtual_page#, chunk_size)*

Operational semantics: For all pages in the chunk, set the *if_hide* and other fields accordingly. Get a new translation table from the permutation unit without performing real permutation and clear the lock bitmap. Later accesses to the chunk must go through the address translation. This instruction can be used when a new memory space is allocated. Since all old contents will be overwritten, no real permutation is needed.

♦*unhide_chunk (begin_virtual_page#)*

Operational semantics: Clear the *if_hide* fields of all pages in the chunk such that the later accesses will go to memory directly. The old contents are discarded.

♦*unlock_block (virtual_page#, start_block_num, num_block)*

Operational semantics: Clear *num_block* consecutive bits in the *lock_bitmap* of a virtual page, starting from *start_block_num*.

Next we discuss some optimizations developed to boost the level of security guarantee offered as well as the performance.

6. HIDE PLUS

This section talks about techniques to achieve a higher level of security guarantee based on the hardware infrastructure and interface proposed earlier. Through compiler analyses and user specifications, we can effectively improve transition coverage and reduce address bus leakage, esp. for sensitive contents with very small overhead.

6.1 Compiler Directed Layout Optimization to Minimize Inter-Chunk Transitions

As mentioned in Section 5, chunk-level protection still exposes inter-chunk transitions since permutations are confined to the blocks only within a chunk. To minimize such exposure, we should put functions that frequently call each other in the same chunk. Similarly, consecutive data accesses that are frequent should be put in the same chunk. We present a compiler optimization that attempts to properly layout code and static data to minimize inter-chunk transitions. This approach is also applicable to heap space that is redistributed to a program – refer to Section 6.2.

For static data, we assume arrays and structures are laid out as a whole. For code, functions are laid out as a whole. Occasionally, we may encounter huge functions[1] or arrays, which can cause big overheads if they are covered with big pages. In such cases we assume that the compiler or the programmer is able to divide them into smaller pieces that are unlikely to transit frequently from one to another. Next we introduce the *transition graph.*

Transition Graph: undirected graph with weighted nodes and edges.

For code, each node represents a function. The weight of the node is the size of the function. Edge weights between nodes represent the call/return frequency between the two functions. If function A calls B or returns from B once, the weight of the edge between node A and B is increased by 1. For static data, each node represents a *data unit*, i.e. a scalar variable, a structure or an array, etc. The weight of the node is the size of the data unit. The weight of the edge is the number of times the two data units are accessed consecutively. For example, if we find a path on which data unit A is accessed immediately after data unit B, the edge weight between them is incremented by 1.

The transition graph is a rough estimation of how frequently nodes transit from one to another, because some transitions can be hidden by the cache. Also, it would be more accurate if we have profile information available. Figure 8 shows the layout algorithm.

[1] In our benchmark, we only observe several functions that exceed page size.

Our goal is to assign nodes to chunks, so that with minimal total *chunk cost* we can *cover* most edge weights. At this point, we need to clarify two things: 1) The total chunk cost is the sum of all chunks' cost, and a chunk's cost is empirically calculated as proportional to its size, since the time to permute grows with the chunk size. 2) If two nodes are in the same chunk, then the edge weight between them is covered.

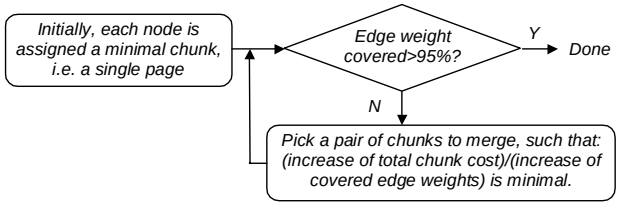


Figure 8. Algorithm to layout code and data.

Initially, each node is assigned to a distinct chunk with minimal size, i.e. a single page. Empirically we loop until over 95% of edge weights are covered. During each iteration, we find a pair of chunks to merge with minimal (increase of total chunk costs)/(increase of covered edge weights) ratio. Notice that, if we merge two chunks, all nodes in the two chunks are now assigned to a new chunk that can hold all nodes and its size is minimal.

6.2 Other Support for Managing Stack and Heap

Stack and heap are dynamically managed memory spaces that must be tackled at runtime. We now discuss several optimizations for protecting stack and heap accesses.

Since stack size is typically small and most activities occur at the top of the stack, we should always try to put the top of the stack inside a chunk boundary. To avoid the exposure of accesses to the top of the stack, the application should check if the callee's frame will cross chunk boundary. If so, it should set the frame pointer to the boundary of the next chunk. Figure 9 shows that when lesser space is available in a chunk than the stack frame size of the callee function, we allocate the callee's stack frame at the start of the next chunk. In this way, we can avoid callee's stack frame from crossing the chunk boundary which might lead to information leakage. Since the parameters have to be passed from caller to callee, this might still lead to some inter-chunk leakage. We suggest parameters are put on the callee's chunk. Although this might still cause several inter-chunk transitions at the beginning of a call, we believe the amount of leakage introduced should be small. On the other hand, the accesses within a stack frame are more frequent and thus, we choose the above solution. In cases where there are accesses across stack frames (such as indirect reference through a pointer to caller's data etc.), we attempt to put the two functions' stack frames in the same chunk.

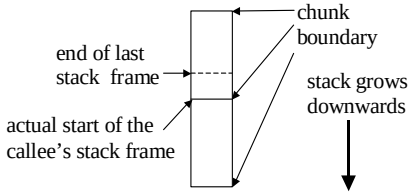


Figure 9. Example for stack.

Since the location and size of the chunk as well as the end of the stack frame are all known at runtime, the compiler simply inserts comparison instructions at the function call site and advances the stack frame accordingly. The new chunk is protected with the *hide_chunk* instruction. Given the stack frame size is typically small, space loss due to this is negligible compared with all memory space taken by a program.

On the other hand, heaps can be huge; therefore more specifications by the programmer are desirable. Next we give the *unlock rule*, which has been found very useful to reduce the number of locked blocks and save unnecessary permutations.

Unlock Rule: If a block will not be referenced after certain point and it is not dirty in the cache, we can directly unlock it.

If a block will no longer be referenced and is not dirty, there is no need to lock it in cache, because its addresses will not appear on the bus any more. The unlock rule suggests that data can be unlocked in the cache as soon as we are sure it will not be referenced later. Especially when only a few blocks in a chunk are locked in the cache, the controller will be reluctant to force a permutation of the entire chunk. Thus, unlocking these blocks without incurring permutation is most desired in this situation.

To unlock a set of blocks, we can use the *unlock_block* instruction. This instruction simply clears the lock bits for those blocks if they are not dirty. *unlock_block* instruction can be put together with code by the compiler or the programmer after data values are determined to be dead through offline analysis. Some heap allocation schemes grab large pieces of memory from heap then redistribute them to the application. Thus, techniques presented in Section 6.1 might be similarly applied to minimize inter-chunk transitions during the above heap allocations.

7. OTHER CONSIDERATIONS

Information leakage prevention is a broad topic, whereas this paper only tackles a particular problem. For instance, system calls can somewhat leak control flow information, however the interaction between application and operating system is unavoidable. This problem is actually left to the programmer to not to put system calls at sensitive points of the code. Also, execution time cannot be hidden due to its tight association with performance. It is normally unreasonable to require the program to run for the same amount of time regardless of inputs. Under our scheme, the attacker observing the address bus still gains some information, such as the moments when permutations take place, the number of accesses between two permutations, etc. However such leakage is much less than that from unprotected address bus. So far we cannot conceive of any valid attacks that might benefit from this type of information leakage.

In a multi-processor system, one block may be present in the caches of other processors, therefore locking and permutation information must be shared and consistent across multiple processors. However the communications among processors are on the bus that is subject to attack. Therefore our scheme cannot work without major modifications. Currently, we regard address bus information leakage prevention in a multiprocessor environment as our future work.

8. EVALUATION AND RESULTS

We evaluate our schemes on a processor model with default parameters in Table 3, in which all 8K chunks are protected. The entire SPEC2000int benchmark suite is used as representative

applications. Implementation is done with the Simplescalar toolset [17] and experiments are based on SimPoint [20]. Each benchmark is fast-forwarded according to SimPoint then simulated by 100M instructions. The 200M Memory bus is 8B wide and fully pipelined.

In Figure 10, we compare IPC, bandwidth usage and transition coverage (i.e. percentage of intra-chunk transitions—Section 5) for 3 models: 1) the one with off-chip shelter buffer in the ORAM paper [9,10]; 2) default model with parameters as listed in Table 3, 8KB chunk, no layout optimizations; 3) a more secure model with 64KB chunk and compiler layout optimizations described in Section 6. For IPC comparison, we normalize all IPC to the original. We also list absolute values of the original IPC. The ORAM model incurs significant slowdown (73%), although we only implemented it at the smallest chunk, i.e. single-page level with 16-block shelter buffer for a fair comparison with the default model. Both default model and the "64K chunk+layout" model shows little slowdown: 0.3% and 1.5% on average.

Table 3. Default architectural parameters.

Clock frequency	1 GHz	L1 I/D	8K DM 1 cycle 32B block
Fetch queue	32 entries	Memory bus	200M, 8 Byte wide
Decode/issue/ commit width	8/8/8	Unified L2	4way, 32B block 1M (12 cycles)
RUU/LSQ size	128/64	Memory latency	80(1^{st}), 5(inter) cycles
TLB miss	30 cycle	Chunk size	8 K, all chunks protected
Permutator	64K	Fetch buffer	8 blocks
Outbuf		Page info caches	8K/1K

Next, we look at the bandwidth usage. We show the percentage of overall bandwidth (i.e. 1.6GB/s) being taken. ORAM uses over 60% of the bandwidth due to its scanning of the whole shelter buffer during each access, the default model and 64K chunk one only use 9% and 15% of the bandwidth. For most benchmarks in SPEC2K, the memory traffic is not a big issue since they take only about 5% of the bandwidth to begin with. To get an idea of the worst-case bandwidth consumption, we reduce the L2 cache size to 512KB and 256KB (due to the nature of SPEC benchmarks, we cannot find one that experiences memory problem with 1MB L2). For 512KB L2, the bandwidth consumption increases by 130% from the default model, whereas 256KB L2 leads to 529% memory traffic increase. As mentioned in earlier sections, the majority of the memory traffic comes from permutations and therefore can be properly pipelined and parallelized with memory banking. For memory-bound applications, it is recommended to use a separate permutation chip to offload the traffic from the front-side bus—Section 4.3.

The bottom graph in Figure 10 gives transition coverage of the 3 models, which is an indication of the level of security guarantee that is achieved (Section 5). The first two models show the same 75% transition coverage on average, because they both protect at 8K chunk level. With 64K chunk and layout optimization, 87% transition coverage is achieved. Several benchmarks get lower transition coverage like gap, vpr mainly because their data layout on the heap is not well organized, which tends to jump among several places with long distances. This also causes more permutations, since the number of blocks each permutation can unlock is lower.

Thus, one can see that a significantly higher transition coverage can be achieved in the 3^{rd} model with almost insignificant performance degradation. The layout optimization provides around 95% transition coverage for code and static data, while it only causes negligible slowdown due to the small size of that part. The rest of the slowdown is due to large chunk-size and overall transition coverage is much higher with little performance penalty.

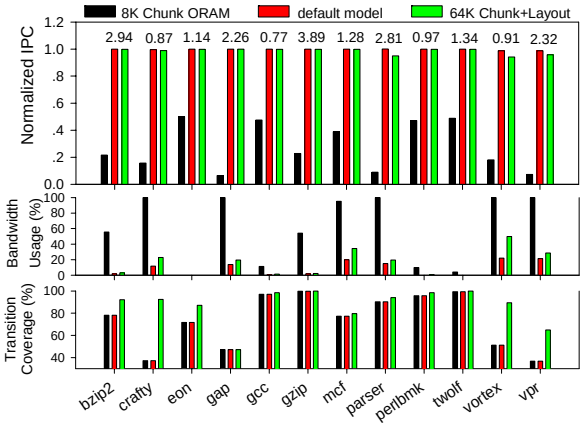


Figure 10. Comparison for 3 representative models.

It is important to understand if the user can provide specifications to exclude a small percentage of code and data as non-security-sensitive, our scheme will eliminate almost all the leakage on the address bus with negligible slowdown. For the default model, the user needs to identify roughly 25% such code and data, while under the "64K Chunk+Layout" model, he only needs to exclude about 5% of the code and static data (Figure 8) and 13% of the data on the heap, assuming stack is protected as in Section 6.2. This means the user specification can be very rough or in same cases, a nice compiler approach will probably do the work too, making our scheme practical to achieve a good level of security guarantee.

In Figure 11, we vary the size of L2 cache under the default model to see how slowdown changes. From Figure 11, only some benchmarks have observable slowdown, typically those with relatively large working sets. Small cache leads to bigger slowdown due to more L2 misses causing more permutations. On average, the slowdown for 512K, 1M, 2M L2 is 1.5%, 0.3%, 0.03% respectively. Our results show that the slowdown is within 20% even for 64K caches, making it applicable to low-end systems with smaller caches where information leakage might be more severe.

In Figure 12, we do sensitivity study for chunk sizes under the default model. All comparisons are against the 8K default model. As protection granularity increases, more slowdown and bandwidth consumption occur. On average, the slowdowns for 16K, 32K, 64K chunk are 0.13%, 0.55%, 1.05% respectively, and the bandwidth increases are 18%, 40%, 73% respectively. Again, a permutation chip could offload this bandwidth increase from front-side bus.

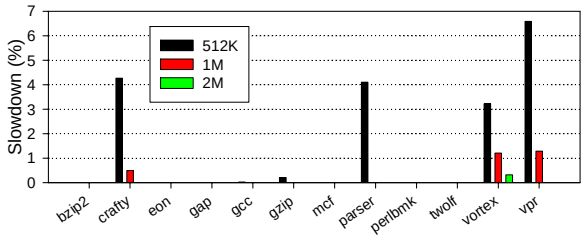


Figure 11. Slowdown comparison for 3 L2 cache sizes.

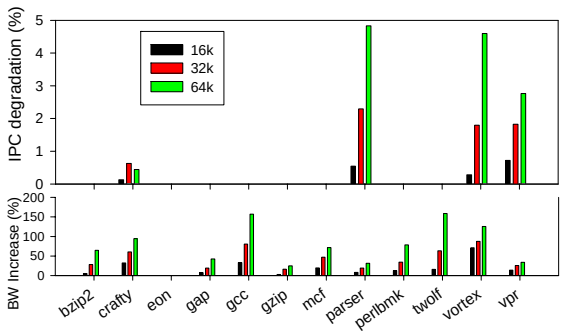


Figure 12. IPC and bandwidth sensitivity to chunk sizes.

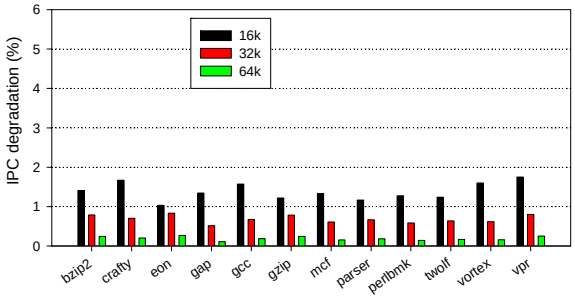


Figure 13.IPC degradation with layout optimization.

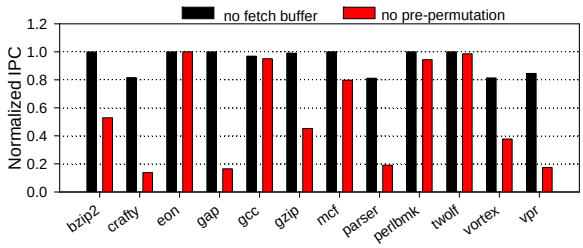


Figure 14. Evaluation of latency hiding techniques.

Figure 13 shows how layout optimization affects the performance. For chunk sizes of 16K, 32K and 64K, the slowdown due to layout optimization is calculated for each benchmark. This figure tells us that with larger chunk size layout optimization causes less slowdown. This is perhaps because with larger chunk size, most code chunks are already covered. In other words, more chunks generated by the layout algorithm are automatically covered without extra performance loss. On average, the slowdowns are 1.38%, 0.68%, 0.19% for 16K, 32K, 64K chunk sizes respectively.

Finally, Figure 14 shows a number of other experiments that slightly change the default model, including 1) remove the fetch buffer; 2) permute when all blocks are locked instead of half way. IPC numbers are normalized to the default model. It suggests the fetch buffer contributes about 6.3% reduction in slowdown whereas pre-permutation contributes 44% on average. Fetch buffer and pre-permutation avoids latencies to be on the critical path. Especially, pre-permutation reduces cascaded permutations when fetches to the same set come before the previous one finishes.

We have also tried to change the L2 cache to 8 way, however it contributed marginally to pre-permutation as 4 way is almost enough. On the contrary, making the L2 8-way can cause more accesses to the same set (due to fewer number of sets). Also, we tried pre-permutation when ¼ of the ways are locked, which shows more traffic due to lesser number of blocks unlocked by each permutation, while changing to ¾ way pre-permutation causes more slowdown due to later permutations causing more stalls. Finally we increase the cache block size to see how slowdown gets affected by it. However it turns out that larger block size has negligible impact on performance.

9. RELATED WORK

It is important to distinguish between "security guarantee" and "seemingly secure". For the latter, there are many ways such as reordering the blocks at runtime, reading blocks to a buffer then writing out after some time to new places, obfuscating the code, issuing random accesses etc.; however all these approaches provide no guarantee on how much information can be leaked. A "seemingly secure" approach should not be taken as serious work in the security domain because it is hard to fathom how powerful (smart) the attacker might be. Only security guarantees establish the security strength of a system without making assumptions about the attacker, which is the essence of this work.

Code obfuscation techniques are only "seemingly secure", but it only makes cracking relatively harder. No security guarantee is provided as such.

The DS5000 series processor supports so-called address bus encryption, which is equivalent to the initial permutation in Figure 5.c. However, it does not permute repeatedly at runtime, therefore the attacker can still construct the CFG in the same way as mentioned in Section 1. The DS5000 also issues random fetches in order to confuse the attacker (seemingly secure). However, random fetches can be easily discerned from true accesses in loops, which repeat more frequently. Actually, DS5002FP has been completely cracked [7].

Goldreich [9,10] proposed three approaches to guarantee no information leakage on the address bus, however all of them can incur big slowdown. For example, the "square-root solution" needs to read the entire shelter buffer before each access; the "hierarchical solution" takes O(t*log(t)*log(t)) memory space after t accesses, causing memory explosion.

The leakage-proof program partitioning work done by Zhang et al. [24,25] tackles a similar problem. Their work focuses on combating control flow information leakage due to the dynamic sequences of program partitions transmitted through network in a networked embedded systems environment. On the other hand, this work focuses on eliminating the control flow information leakage due to code/data blocks transmitted through system address bus. Both the assumption and the solution of their work are fundamentally different from this one.

10. CONCLUSION

In this work, we provide a lightweight solution to the problem of information leakage on the address bus due to both data and code accesses. We show that this problem is critical for XOM-based secure architecture to solve software IP protection issues and stop side channel attacks in encryption based approaches. However, all known solutions with enough security guarantee [10] suffer from very high performance degradation.

In this work, we propose the HIDE infrastructure including the hide cache with block locking and permutation mechanisms.

HIDE provides chunk-level protection and interface for compiler optimizations. Then we propose compiler optimizations for code and data layouts and other runtime optimizations to reduce overheads and improve level of security guarantee.

Our results show with 64K chunk protection and the layout optimization, we can guarantee 87% of the address sequence is protected, in which 95% of the accesses to code and static data are hidden. With the HIDE infrastructure, interfaces are provided for the compiler or user to further improve the level of security guarantee or to narrow down the protection domain to achieve almost complete protection. In this way, all security sensitive code/data could be identified and effectively protected in terms of the leakage on the address bus. The performance overhead is at most 1.5% in our experiments. The increase of the bus traffic takes a very small part of the total bandwidth available in our benchmarks. The majority of the traffic increase is due to permutations. Such traffic is very regular therefore we can reduce its overhead in multiple ways as suggested in the paper. Finally, most on-chip hardware components for HIDE are small. The largest component, i.e. the permutation unit with 64KB *out_buffer* can be shifted to the permutation chip as well. Due to the low overhead of the HIDE infrastructure, it is possible to apply it to low-end systems with smaller cache where leakage on the address bus might be more severe.

11. ACKNOWLEDGEMENTS

This work was supported in part by NSF grants CCR-0220262, CCR-0208953 and CCR-0326396. The authors would like to sincerely thank all reviewers esp. our shepherd Dr. Chandramohan Thekkath (Microsoft Research) for their critical comments and help in revising the manuscript. Also, the authors would like to thank Dr. Hsien-Hsin Lee (ECE Dept., Georgia Tech) for some initial discussions on this topic.

REFERENCES

[1] D.Lie, C.Thekkath, M.Mitchell, P.Lincoln, D.Boneh, J.Mitchell, M.Horowitz, "Architectural Support for Copy and Tamper Resistant Software," *International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[2] D.Lie, C.Thekkath, M.Horowitz, "Implementing an Untrusted Operating System on Trusted Hardware," *19th ACM Symposium on Operating Systems Principles (SOSP 2003)*, Oct. 2003.

[3] J.Yang, Y.Zhang, L.Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," *International Symposium on Microarchitecture*, Dec. 2003.

[4] E.Suh, D.Clarke, B.Gassend, M.v.Dijk, S.Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors", *International Symposium on Microarchitecture*, Dec. 2003.

[5] B.Gassend, G.E.Suh, D.Clarke, M.v.Dijk, S.Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification", *International Symposium on High Performance Computer Architecture*, Feb. 2003.

[6] G.E.Suh, D.Clarke, B.Gassend, M.v.Dijk, S.Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," *International Conference on Supercomputing*, Jun. 2003.

[7] M.G.Kuhn, "Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP," *IEEE Transaction on Computers*, Vol.47, No.10, pp.1153-1157, 1998.

[8] M.Kuhn, "The TrustNo 1 Cryptoprocessor Concept," *CS555 Report*, Purdue Univ. 1997.

[9] O.Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMs," *The 19th Annual ACM Symposium on Theory of Computing (STOC)*, 1987.

[10] O.Goldreich, R. Ostrovsky, "Software Protection and Simulation on Oblivious RAMs," *Journal of the ACM*, Vol.43, No.3, 1996.

[11] "DS5002FP secure microprocessor chip data sheet," *Dallas Semiconductor*.

[12] J.R.Ullman, "An Algorithm for subgraph Isomorphism," *Journal of the ACM*, Vol.23, pp.31-42, 1976.

[13] VFLib Graph Matching Library, http://amalfi.dis.unina.it/graph/db/vflib-2.0/doc/vflib-1.html

[14] P.C.Kocher, "Timing attacks on implementations of Die-Hellman, RSA, DSS, and other systems," *International Cryptology Conference*, 1996.

[15] P.Kocher, J.Jaffe, B.Jun, "Differential Power Analysis", *International Cryptology Conference*, 1999.

[16] K.Gandolfi, C.Mourtel, F.Olivier, "Electromagnetic Analysis: Concrete Cresults," *In Workshop on Cryptographic hardware and Embedded Systems*, 2001.

[17] D.Burger, T.M.Austin. "The SimpleScalar Tool Set Version 2.0," *TR. 1342*, Univ. of Wisconsin--Madison, May 1997.

[18] R.Anderson, M.Kuhn, "Low Cost Attacks on Tamper Resistant Devices," *Security Protocols Workshop*, 1997.

[19] J. Kelsey, B. Schneier, D.Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *European Symposium on Research in Computer Security*, Sep. 1998

[20] T.Sherwood, E.Perelman, G.Hamerly, B.Calder, "Automatically Characterizing Large Scale Program Behavior," *International Conference on Architectural Support for Programming Languages and Operating Systems* Oct. 2002.

[21] A.Huang, "Keeping Secrets in Hardware: the Microsoft Xbox (TM) Case Study," *MIT TR. AIM-2002-008*, May 26, 2002.

[22] C. McClure, "Software Reuse Planning by Way of Domain Analysis," Technical Paper, *Extended Intelligence, Inc.* http://www.reusability.com.

[23] D.E.Knuth, "Seminumerical Algorithms," *The Art of Computer Programming*, Vol. 3, Addison Wesley 1981.

[24] T.Zhang, S.Pande, A.D.Santos, F.Bruecklmayer, "Leakage-proof Program Partitioning," *International Conference on Compiler, Architecture and Synthesis for Embedded Systems*, Oct. 2002.

[25] T.Zhang, S.Pande, A.Valverde, "Tamper-resistant Whole Program Partitioning," *International Conference on Languages, Compilers, and Tools for Embedded Systems*, Jun. 2003.